Timing prediction for service-based applications mapped on Linux-based multi-core platforms

Ruben Jonk*, Jeroen Voeten†*, Marc Geilen*, Twan Basten*† and Ramon Schiffelers‡*
*Eindhoven University of Technology
Eindhoven, The Netherlands
{r.j.w.jonk, j.p.m.voeten, m.c.w.geilen, a.a.basten}@tue.nl
†ESI, TNO
Eindhoven, The Netherlands
‡ASML
Veldhoven, The Netherlands
ramon.schiffelers@asml.com

***Abstract*—We develop a model-based approach to predict timing of service-based software applications on Linux-based multi-core platforms for alternative mappings (affinity and priority settings). Service-based applications consist of communicating sequential (Linux) processes. These processes execute functions (also called services), but can only execute them one at a time. Models are inferred automatically from execution traces to enable timing optimization of existing (legacy) systems. Our approach relies on a linear progress approximation of functions. We compute the expected share of each function based on the mapping (affinity and priority) parameters and the functions that are currently active. We validate our models by carrying out a controlled lab experiment consisting of a multi-process pipelined application mapped in different ways on a quadcore Intel i7 processor. A broad class of affinity and priority settings is fundamentally unpredictable due to Linux binding policies. We show that predictability can be achieved if the platform is partitioned in disjoint clusters of cores such that i) each process is bound to such a cluster, ii) processes with non real-time priorities are bound to singleton clusters, and iii) all processes bound to a non-singleton cluster have different real-time priorities. For mappings using singleton clusters with niceness priorities only, our model predicts execution latencies (for each pipeline iteration) with errors less than 5% relative to the measured execution times. For mappings using a non-singleton cluster (with different real-time priorities) relative errors of less than 2% are obtained. When real-time and niceness priorities are mixed, we predict with errors of 7%.**

***Keywords*-Linux multi-core platform; Fair Share Scheduling; Software-Oriented Architecture; Y-chart model**

I. Introduction

The work in this paper is inspired by the performance challenges of the lithography systems of ASML. Lithography systems are cyber-physical systems in which the execution of physical actions and software actions are tightly intertwined. Physical components perform physical actions concerning for instance the movement of a robot. Software components execute software actions to control these physical actions or to correct for physical disturbances. Throughput is one of the key performance indicators of these systems. A key architectural principle to optimize throughput is that it should be limited by physics only, because improvements there are far more costly than in the compute platform. This means that software actions should be executed outside of the critical path, which consists of physical actions. Another key performance indicator is system accuracy. To drive this performance indicator, an increasing number of software actions in the form of correction algorithms, models, and control loops are added and these actions have to be computed in less time to drive system throughput. For reasons of cost, ease of programming and maintainability, these software actions are executed on standard off-the-shelf multi-processor multi-core platforms making their execution time variable and hard to predict. The increasing complexity (i.e. increasing number of software actions, decreasing timing budgets and increasing multiplexing on platform resources) puts an increasing amount of stress on the key architectural principle, i.e. it is ever more challenging to keep the software from the critical path.

ASML employs a service-based software architecture [16] containing hundreds of communicating processes. Many of these processes are mapped on a multi-core platform running Linux using Real-Time Scheduling (RTS) in combination with Completely Fair Scheduling (CFS) [12]. The mapping of the application on the platform is determined by two parameters. The *niceness* values given to the processes determine the core share each process obtains at any moment in time. The processor *affinity* specifies the collection of cores on which a particular service is allowed to run.

The main long-term challenge is to automatically map the application in such a way that system throughput is optimized. As a first step we develop in this paper a model-based approach to predict timing performance for alternative mappings on the platform. Models are inferred automatically from execution traces, enabling timing optimization of existing systems. The focus is on predicting latencies, in particular the makespan of the execution of an application

and the completion times of each individual application task. The novelty in our work is four-fold: i) we develop a Y-chart-based framework for service-based software architectures including a two-step mapping scheme in which tasks are mapped to processes and processes to the multi-core platform, ii) we infer from execution traces a sound and complete task dependency graph of the application to support performance prediction for alternative mappings, iii) we establish linear progress approximation of tasks, abstracting the Linux RTS and CFS implementations, for efficient schedule prediction, and iv) we establish explicit Linux mapping conditions that lead to timing predictability.

The paper is organized as follows. In Sections II the basic working of the Linux scheduler is explained. In Section III we establish the mapping conditions that result in timing predictability. In Section IV we introduce the modelling approach and in Section V the transformations applied on the models. In Section VI we validate the predictive power of the model through a set of lab experiments. In Section VII we provide an overview of related work. Section VIII concludes.

II. Linux scheduling

The Linux operating system combines two scheduling policies, namely the Completely Fair Scheduler (CFS) policy and the Real-Time Scheduler (RTS) policy. Tasks scheduled by the RTS policy have higher priorities than tasks scheduled by the CFS policy. Within the RTS policy, a task with a higher priority level preempts tasks with lower priority levels, whereas tasks of equal priority are executed in a preemptive round-robin fashion. The CFS policy is not preemptive, but emulates an ideal fair share multi-processor. CFS does not use a time wheel like other time division multiplexing schedulers, but instead computes which of the processes has received the least amount of processing time and schedules it on the core. This fair share distribution is limited to processes running on a given core, rather than the processes on the whole platform. For the CFS policy, the share a process receives from a certain core is determined by the niceness values of the processes bound to this core. The niceness of a process determines the weight of the process where a lower niceness leads to a higher weight. The weights are determine given by $w = \frac{1024}{1.25^n}$, where n is the niceness and w is the weight. The weight is a key parameter that determines the share the process receives from the OS.

III. Time Predictable Linux Scheduling

In this section, we identify under which mappings we can predict the execution times of functions being executed by the processes running on the platform. The Linux OS binds processes to cores according to their affinity sets. This binding step is non-deterministic, thus leading to unpredictable timing behaviour of processes. To obtain predictable timing, we therefore have to prevent the OS from making such non-deterministic binding decisions. This can be achieved by

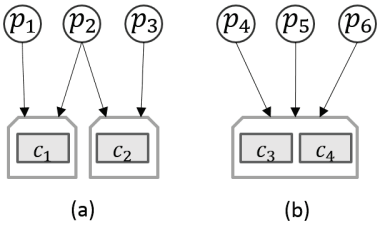

Figure 1: An illustration of the unpredictable mappings.

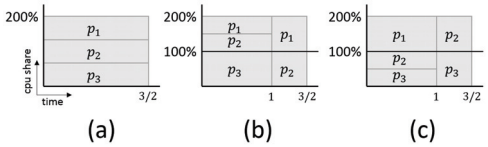

Figure 2: The (a) fair share Gantt chart and (b) and (c) the possible Gantt charts of a concrete run from the mapping in Figure 1(a).

partitioning the platform in disjoint clusters of cores such that i) each process is bound to such a cluster, ii) processes with non real-time priorities are bound to singleton clusters, and iii) all processes bound to a non-singleton cluster have different real-time priorities.

To illustrate timing unpredictability for mappings not adhering to the three constraints, Figure 1 shows a mapping scheme of processes p_1 through p_6 on the cores c_1 through c_4 divided in four sets of cores, $\{c_1\}$, $\{c_1, c_2\}$, $\{c_2\}$, and $\{c_3, c_4\}$. In Figure 1(a), processes p_1, p_2 and p_3 are mapped to $\{c_1\}$, $\{c_1, c_2\}$ and $\{c_2\}$ respectively. Notice that this mapping violates condition (i) of the constraints as the clusters of cores do not form a partition. Figure 2(a) shows the execution Gantt chart for the mapping from Figure 1(a) in case the workload and priorities are identical assuming the OS would use a global fair-share policy where each task executes in $3/2$ time units. This fair behaviour does not occur in practice though, as the OS makes a decision to bind p_2 either to c_1 or to c_2. As a result, the execution times of p_1 and p_3 vary significantly, depending on the (non-deterministic) binding decision made. The resulting Gantt charts are depicted in Figure 2(b) (p_2 bound to c_1) and (c) (p_2 bound to c_2).

To illustrate the rational for conditions (ii) and (iii), Figure 1(b) shows p_4, p_5 and p_6 all mapped to $\{c_3, c_4\}$. In this case, assuming that the priorities are non-real-time (violating constraint (ii)) or assuming that are all have equal real-time priorities (violating constraint (iii)), the OS can bind the processes in different ways (in a similar manner as described before) leading to similar variations in Gantt charts.

In the model-based approach described in the next section, these constraints are formalized. They are further satisfied by the case studies in Section VI.

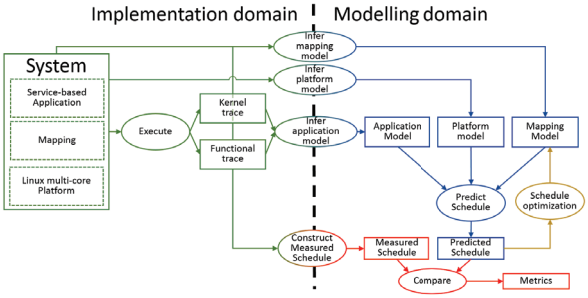

Figure 3: An overview of the approach

IV. Model-based approach

The modelling approach to predict timing performance of service-based software applications for alternative mappings (affinity and priority settings) on Linux-based multi-core platforms is depicted in Figure 3. The rectangles in the figure denote data artifacts, the ellipses refer to data transformations and the arrows denote the direction of the data flow. The next section gives a brief overview of the approach, while the different elements are explained in detail in the corresponding subsections. The transformations between data artifacts are discussed in Section V.

A. Overview of the approach

The approach distinguishes between the implementation domain and the modeling domain. The *implementation domain* is concerned with the service-based application software mapped onto a Linux-based multi-core platform. A service-based application [16] consists of communicating sequential processes. These processes execute functions (also called services), but can only execute them one at a time. Processes communicate through remote function calls, where the call may be followed by a reply. When a call cannot be dealt with immediately, it will be queued. Processes serve enqueued function calls in a first-come-first-served fashion. These processes are executed by the platform. Upon *execution*, they produce a *functional trace* and a *kernel trace*. A functional trace contains information about the functions executed by the different processes and their call and reply dependencies. A kernel trace contains detailed information about the time spent by each process on each of the core cores of the multi-core platform.

The *modelling domain* is concerned with the mathematical model of the system. This model follows the well-known Y-chart approach [9], explicitly distinguishing *application*, *platform* and *mapping* models. An application model is a weighted directed acyclic graph containing tasks and dependencies, where the tasks are annotated with nominal execution times (the total time a process needs to be scheduled on the core to execute the task) and dependencies by communication times. Tasks and dependencies in the modeling domain correspond to (parts of) functions and communications in the implementation domain. A platform model reflects the number of available cores. A mapping model is threefold. First it maps tasks to processes, secondly it binds processes to the cores they are allowed to run on and thirdly it assigns priorities to these processes.

Application models are automatically inferred from functional traces and kernel traces. Platform models are inferred from static system information. To infer mapping models, both static system information and functional traces are required.

Together, application, platform and mapping models contain sufficient information to *predict* a *schedule*. A *schedule* consists of the starting and finishing times of each task in the application model.

To validate the accuracy of a prediction, the *predicted schedule* can be *compared* to a *measured schedule*. A measured schedule is derived directly from a functional trace. The comparison uses distance *metrics* to quantify the difference between predicted and measured schedules.

Once a model has been validated to yield accurate predictions, it can be used for *schedule optimization* by exploring alternative mappings. For this purpose the traces from the actual system are not required anymore. In the following subsections the individual steps in the approach are elaborated.

B. System and execution traces

1) The system: Figure 4(a) shows a service-based application mapped on a Linux-based dual-core platform, which we will use as a running example. The application consists of four processes named P_0, P_1, P_2 and P_3. Process P_0 can execute function f_0, P_1 can execute f_1, P_2 can execute f_2 and P_3 can execute functions f_3 and g_3. The execution starts with f_0. This execution results in remote function calls of f_1, f_2 and f_3. This is denoted in the figure by the single-headed arrows. In the call to f_3, f_0 expects a reply, which is denoted by the circle on the start of the arrow. Remote function calls and replies are communicated on TCP connections.

Processes are mapped on the dual-core platform with cores c_1 and c_2. As shown by the dashed arrows, process P_0 runs on core c_2 and the other processes run on core c_1. Processes are given priorities, indicated by the labels next to the dashed arrows. Process P_3 has priority 50 referring to a real-time Linux priority. Processes P_0, P_1 and P_2 have priorities 120, 123 and 120 respectively. These priorities are niceness priorities in Linux, corresponding to niceness levels 0, 3 and 0 respectively at the user-level of Linux.

2) Functional Traces: Figure 4(b) shows a functional trace corresponding to an execution of the system depicted in Figure 4(a). On the vertical axis the processes are shown and on the horizontal axis time is represented. The execution of the different functions are shown by the solid rectangles. In addition the internal function call-stack representing remote function calls and the reception of replies are shown

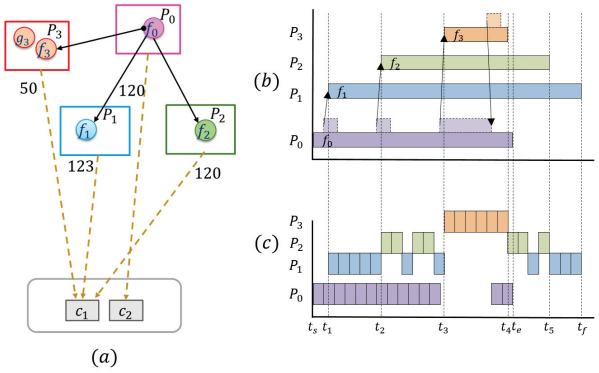

Figure 4: (a) Schematic overview of an example application and; (b) functional and (c) kernel trace matching the application in (a).

(denoted by dashed rectangles) next to the remote call and reply communications (denoted by the arrows).

3) Kernel Traces: A solid rectangle in Figure 4(b) shows that a process is occupied executing a function. For instance during the time interval $[t_s, t_e]$ process P_0 is executing function f_0. This does not imply that this process is active on the platform during the complete time interval. This is shown in Figure 4(c). This figure visualizes a kernel trace obtained from LTTng [3] and shows when each process is using a time slice on one of the cores of the platform. From the figure it is clear that during the execution of f_3, P_0 is idle waiting for a reply from P_3. The figure also shows that during interval $[t_2, t_3]$, processes P_1 and P_2 are using c_1 alternatingly, where P_2 obtains twice as much share as P_1 (in correspondence to the specified niceness values in Figure 4(a)). During interval $[t_3, t_4]$ processes P_1 and P_2 are preempted by process P_3 due to its specified real-time priority and due to the fact that these processes are all bound to core c_1.

C. Application model

An application model G is a weighted directed acyclic graph:

$$G = (T, \rightarrow, E, D), \tag{1}$$

where T is a set of tasks and $\rightarrow \subseteq T \times T$ represents the dependencies between the tasks. Tasks and dependencies are mathematical representations of the functions and remote function calls measured in a functional trace. Internal function calls (denoted by the dashed rectangles in Figure 4(b)) are abstracted from. We assume the graph to contain one source, that is, exactly one task without any incoming dependencies. This task corresponds to the main function of the application, from which the other functions are (remotely) called. $E : T \mapsto \mathbb{R}_0^+$ maps tasks to nominal (non-negative) execution times which are obtained from a kernel trace. $D : \rightarrow \mapsto \mathbb{R}_0^+$ maps dependencies to (non-negative) communication times. The interpretation of these communication times is that the enabling time of a task equals the maximum of the finishing times of all predecessor tasks plus their corresponding communication times. Dependencies between tasks that are mapped to the same processes are assigned communication time zero.

D. Platform Model

A platform model that is sufficient for our purposes is a set of resources R representing the cores. The model assumes a homogeneous system, i.e., the capacities of all the resources are equal and that cache behaviour is included in the time a process spends on the CPU whilst executing a task.

E. Mapping Model

A mapping model M for a set T of tasks and a set R of resources is given by

$$M = (P, \Pi, A, N), \tag{2}$$

where P is a set of processes and $\Pi : T \mapsto P$ maps each task to a process. Tasks executed by processes correspond to functions executed by the processes measured in a functional trace. A is a mapping from processes to a set resources they are allowed to run on: $A : P \mapsto \mathcal{P}(R) \backslash \emptyset$. A captures the affinity settings of Linux processes. Furthermore, each process is mapped to a priority level $N : P \mapsto [1, 139]$. The priority interval $[1, 99]$ indicates real-time priorities and the interval $[100, 139]$ refers to niceness values, in correspondence to process priority settings in Linux. Note that Linux user settings of niceness values fall in the range $[-20, 19]$. Internally these values are converted by adding the number 120.

Notice that unlike the traditional Y-chart approach [9], our mapping model has two layers. Application tasks are mapped to processes, and these processes in turn are bound to resources in the platform model. In fact processes can be considered to be resources themselves. The reason is that they represent the main concepts in a service-based software architecture [16], which are only able to execute one task at a time. We elaborate on this further in the related research, Section VII.

F. Schedule

We define a schedule σ as follows:

$$\sigma = (st, ft), \tag{3}$$

where $st : T \mapsto \mathbb{R}_0^+$ and $ft : T \mapsto \mathbb{R}_0^+$ map each task to its starting and finishing times, respectively.

For a schedule to be valid, tasks may not start before their preceding tasks are finished and that tasks may not finish before they are started. Therefore, the following two properties must hold in any valid schedule:

(i) $\forall_{u \rightarrow v}\ ft(u) + D(u \rightarrow v) \leq st(v)$
(ii) $\forall_{t \in T}\ st(t) \leq ft(t)$

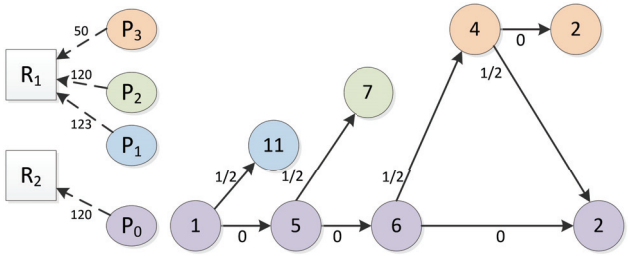

Figure 5: The model extracted from the traces and static information.

G. Predictability Constraints

Not every mapping leads to an accurately predictable schedule as discussed in Section II. A mapping (P, Π, A, N) is predictable if there exists a partition $\mathcal{R}$ of the set of resources R such that the following conditions are satisfied:

(i) For all $p \in P$, $A(p) \in \mathcal{R}$.
(ii) For all $p \in P$, if $N(p) \geq 100$ then $|A(p)| = 1$.
(iii) For all $p_1, p_2 \in P$, if $p_1 \neq p_2$ and $A(p_1) = A(p_2)$ and $|A(p_1)| > 1$ then $N(p_1) \neq N(p_2)$.

$\mathcal{R}$ partitions the set of all resources in disjoint subsets which we call clusters. Condition (i) states that each process is mapped to such a cluster. Hence if two processes share a resource, their affinity set must be the same. Condition (ii) states that only processes with real-time priorities can be bound to a cluster with more than one resource. Hence non real-time processes must be bound to clusters of size 1. Condition (iii) states that for any cluster with more than one resource, the real-time priorities of the processes bound to this cluster must be different. Notice that the processes bound to a cluster of size one can have mixed real-time and non real-time priorities. In this case the real-time priorities can be used multiple times.

V. Model Transformations

This section discusses the model transformations applied on the models in the workflow depicted in Figure 3.

A. Model inference

Model inference is concerned with the transformation and combination into a model of a functional trace, a kernel trace and static system information, into a model. Figure 5 shows the task graph extracted from the functional trace shown in Figure 4(b) and annotated with the information from the kernel trace in Figure 4(c). In addition static information (see Figure 4(a)) is used to infer the platform and mapping model. The transformation is performed in three phases.

(1) In the first phase the functional trace is used to construct a task graph. In this phase we identify the tasks (T), their dependencies ($\rightarrow$), the processes (P) and the mapping of tasks to these processes (Π). In Figure 5 set T is visualized by the circles, $\rightarrow$ by the arrows between tasks, C by the ellipses and c by the colours and rows of the tasks. The functional trace for process P_0 in Figure 4(a) contains three remote function calls. At each of the starting points of the communication arrows, the functional trace is split in separate segments to ensure that each remote function call starts at the appropriate time during the execution of f_0 on process p_0. As a result we obtain four individual tasks, which are shown on the bottom row in Figure 5. Upon the third call, function f_3 generates a reply. Therefore we split this function in two distinct segments, resulting in two corresponding tasks in Figure 5. Task dependencies are determined by functional dependencies in the functional trace. A dependency between tasks is added when a dependency between the corresponding parts exists in the functional trace, or between tasks that originate from adjacent segments of a function call. After the task graph is constructed, each task is mapped to the process in correspondence to the related function and process in the functional trace.

(2) In the second phase the functional trace and kernel trace are used to annotate the tasks and dependencies with execution times and communication times. The starting and finishing times of each segment of the functional trace are used to compute the communication times $D(t)$ present in the platform (D in the application model). From all computed communication times, a representative value of the system is computed for the system based on the median value of all the communication times. These are shown as labels on the dependencies in Figure 5. Nominal execution times of tasks (E in the application model), denoted by the task labels in terms of time units of exactly one time slice in Figure 5 are obtained from the measured on-core times in the kernel trace. These correspond to the accumulated time-slices the process used to execute the segment (see Figure 4(c)). This accumulated time includes the time a process is waiting for the cache. Depending on the whether cache interaction is a major factor in a given application, in the future the model can be extended with an application slowdown factor as described by Subramanian et al. [15].

(3) In the third phase the static information depicted in Figure 4(a) is used to infer the platform model (R) and the mapping of processes to the platform (affinity A and priority N). The mapping is visualized by the labeled dashed arrows in Figure 5. Here process P_0 is bound to resource R_2 with priority 120. Processes P_1, P_2 and P_3 are bound to resource R_1 with priorities 123, 120 and 50 respectively.

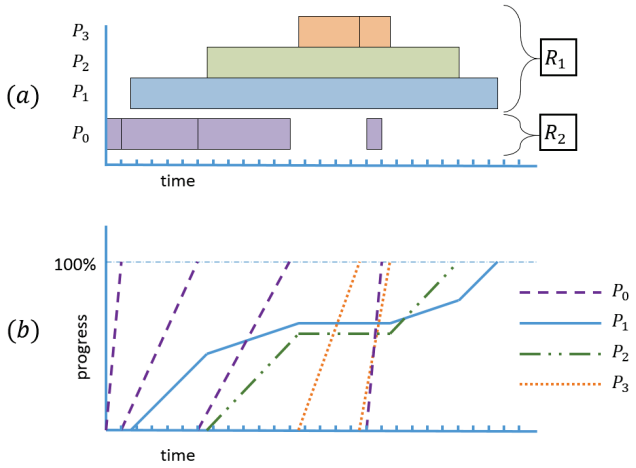

Figure 6: (a) A schedule and (b) a linear progress approximation for the tasks running on R_1 and R_2.

B. Schedule Prediction

We develop a model of execution, in accordance to the combined Linux CFS and RTS policies, to predict schedules based on application, platform and mapping models. The model of execution encompasses both real-time and niceness priorities. Processes with real-time priorities, which execute in a round-robin fashion, preempt processes with lower real-time priorities as well as processes with niceness priorities, which are multiplexed using the Completely Fair Scheduler.

Schedule prediction is done by approximating the scheduler by a piecewise linear progress function of each task. The preemptive property of real-time tasks together with weight-based fair shares allow us to make an approximation of the progress of tasks based on their total required time spent on the resources, and the tasks currently active on the resources.

A schematic view of the approach taken is shown in Figure 6(b) which depicts the processes running on resources R_1 and R_2. In Figure 6(a), the four processes executing tasks on the resources R_1 and R_2 are shown. Initially only one task is active on R_2. After it finishes, it starts the task handled by process P_1 and process P_0 continues with the next task. Five time units later, the second task on P_0 is completed and it starts the task handled by P_2. The third task is then introduced on R_1 after six more time units. It returns a reply after four time units to process P_0, which can then continue with its next task. In Figure 6(b) the linear approximation of the progress of each process executing its task. The progress of each process is represented by a connected piecewise linear function, with slope changes occurring upon each change in the set of active processes. The slope of each process depends on the total load of the task, the priority of the process that executes the task and the other active processes with their respective priorities. For example, initially P_1 executes the task with rate $\frac{1}{11}$, whereas it proceeds with rate $\frac{1}{33}$ in the second segment when P_2 starts executing its task. P_1 and P_2 are halted when P_3 starts executing its task, as depicted by the rate of zero. The starting and finishing times as computed in this manner as shown in Figure 6(b) are computed from the execution model. They are used to construct a schedule σ from the model, depicted in Figure 6(a) for this example.

As shown in the example, we compute the resource share each process receives at any given time. Furthermore, we know that these shares can only change upon the termination or start of a task. Therefore our algorithm identifies the discrete time steps at which tasks terminate or start and then computes the changes in resource shares. Processes can only handle one task at a time, so a FIFO queue of enabled tasks is maintained for each process to store tasks which can start execution as soon as the process is (or becomes) idle. Due to the single-threaded nature of each process, we treat each process as a resource that is required to be claimed prior to execution. Once an enabled task has claimed the process, the task becomes an active task. In order to ensure a function call is completely finished prior to handling a new function call, the tasks originating from the same function have to execute uninterrupted, even if the process is idle while waiting for a reply. To this end, the claim on the process resource is transferred from one task to the next one until the last task in such a sequence, which then releases the claim. At each identified time step, five algorithmic phases are performed, starting at time $\tau = 0$ with the active task source(G):

(1) For each unclaimed process, take the first of the enqueued tasks, if available. This task claims the process and is added to the active tasks and it is assigned a starting time τ. Terminate the algorithm if there are no remaining active tasks.

(2) For each cluster R_c, compute the fractions of the resources for tasks T_c mapped onto the cluster as shown in Algorithm 1. First the set of rates (fraction of the compute resource) F and the size n of cluster R_c are initialized. Then the real-time priority levels starting with the highest priority are traversed. If there are less tasks for a priority level (lines 4-8) than there are cores left in the cluster, each task receives a full share of 1 and the number of available cores is reduced by the number of considered tasks. Otherwise (lines 9-13), the remaining tasks receive an equal share of the remaining core and the algorithm returns the set F (line 13). Due to the constraints outlined in the previous section, lines 9-13 are only executed with $n = 1$. All tasks with a niceness priority are then scheduled only if there are no real-time priorities, as a consequence of the constraints in Section IV-G. These tasks receive a rate according to their own weight compared to the total weight of all tasks on the singleton cluster (lines 16-20), and the set

F is returned on line 22.

(3) Compute the minimal time-step τ_s required to finish a task or to activate a new task (i.e. after the communication time between a new task and its predecessor has passed). $\tau_t = min_{t \in T_c} \frac{E(t)}{f(t)}$, $\tau_d = min_{d \in D_c} D(t)$, $\tau_s = min(\tau_t, \tau_d)$.

(4) Progress each active task and dependency according to the rates F and time-step τ_s. For each active task t which receives a fraction $0 < f(t) \leq 1$ of the resource, compute the remaining execution time $E(t) = E(t) - f(t) \times \tau_s$ and for each dependency d compute $D(d) = D(d) - \tau_s$. Increase the elapsed time: $\tau = \tau + \tau_s$.

(5) For each task t with $E(t) = 0$, assign it the current time τ as finishing time. Each dependency originating from t is added to the set of active dependencies. If one of the successors of t uses the same process (i.e. they originate from the same function), transfer the claim of the process to this task, otherwise release the process. For each dependency d with $D(d) = 0$, add the target task of d to the queue of its process resource, or to the active tasks if the task already has a claim on the resource. Return to phase 1.

After termination of the algorithm, each task in G has received a starting time st and a finishing time ft, and is stored in a schedule σ.

Algorithm 1 Compute fractions

Require: Cluster R_c, active tasks T_c
Ensure: Set F of fractions f : $T \mapsto \mathbb{R}_0^+$
  initialization: $F \leftarrow \emptyset$; $n = |R_c|$
  for $p = 99$ to 0 **do**
    $T_p = \{t \in T_c \mid N(t) = p\}$
    if $|T_p| < n$ **then**
      $n = n - |T_p|$
      for all $t \in T_p$ **do**
        $F \leftarrow F \cup (t, 1)$
      end for
    else
      for all $t \in T_p$ **do**
        $F \leftarrow F \cup (t, \frac{1}{|T_p|})$
      end for
      return F
    end if
  end for
  $T_n \leftarrow \{t \in T_c \mid N(t) > 100\}$
  $W = \sum_{t \in T_n} \frac{1024}{1.25^{(N(t)-120)}}$
  for $t \in T_n$ **do**
    $w = \frac{1024}{1.25^{(N(t)-120)}}$
    $F \leftarrow F \cup (t, w/W)$
  end for
  return F

C. Schedule Comparison

Our goal is to assess the predictive power of the model by comparing the predicted schedule from the model, with the measured schedule. This is accomplished by defining metrics which relate the starting and finishing times of both the measured and predicted schedules. For each use case, the metric(s) used to quantify an accurate prediction differ and any metrics quantifying the accuracy of the model are therefore tailored to the usecase. The metrics used in this paper to validate the model for our usecase are discussed in Section VI.

VI. Case study

A usecase of this model is to predict the performance of a pipelined application. Pipelined applications are common in systems controlling production machines, such as the lithography machines produced by ASML.

In this case study we use a fictional application inspired from the domain of lithography systems. In this application, we mimic the process of computing setpoints for handling the wafer scanning stage. This stage is a series of scan segments, each preceded by computing the setpoint for the next scan action. Each iteration of the pipelined execution resembles a new instance of a scan segment in the machine, e.g. computing subsequent setpoints for the scanner. Figure 7 shows a schematic overview of the application task graph. Each row corresponds to a process and each colour indicates the degree of parallelism of the application, i.e. per colour only one task can be active at any given time, and all subsequent tasks executed by a process must be finished before a new iteration may start on the corresponding process. The labels inside the tasks denote the number of execution time units of the task. In our experiments, we vary the length of this time unit.

The processes from the task graph (annoted with nominal execution times and the binding of tasks to processes) in Figure 7 are mapped onto a platform consisting of three cores using three different mapping schemes, as shown in Figure 8. In the real-time mapping case, each of the twelve processes are given a unique real-time priority and is bound to the cluster consisting of all three cores. In the singleton mapping each process is bound to a single core with a niceness priority (denoted by the labels on the dashed arrows). In the combined mapping four processes are bound to one core using niceness priorities, and the other eight to the remaining cores using unique real-time priorities.

We implement the pipelined application in the C programming language using the *rpcgen 2.19* tool [1] that generates middleware code to perform remote procedure calls. We instrument the code with tracing statements to trace every function call. Upon completion of a measurement, the trace is written to file and used for analysis. We execute the application on a quadcore Intel i7 processor running Linux version 3.13.0-59-generic operating system

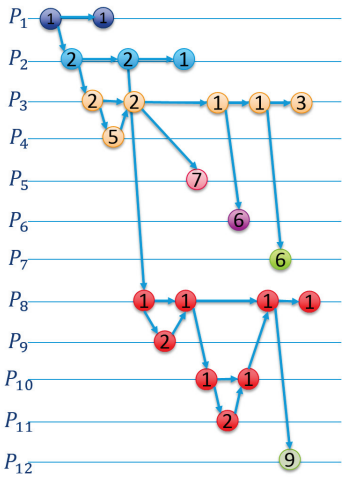

Figure 7: The task graph designed for the case study.

with hyperthreading disabled. Kernel tracing is performed using *LTTng 2.9.5*.

We are interested in predicting latency properties of the application, implying that we want to predict the time between the start of an iteration and the completion time of this iteration. We are also interested in the total makespan of the application.

We identify two metrics that are of interest for the use-case: (1) the latency distance between each individual iteration of the pipeline, and (2) the total distance in makespan. Let σ_m be the schedule obtained from the measured application and let σ_p be the schedule obtained from the model using the same mapping. Metrics (1) and (2) are formally defined as follows.

(1) The latency $l_\epsilon(i)$ of iteration i of schedule σ_ϵ is given by $l_\epsilon(i) = \max_j(ft_\epsilon(t_j^i)) - st_\epsilon(t_0^i)$ where t_j^i denotes task t_j in a given iteration i, where t_0 is the first task of the iteration. The equation to compute the latency distance is then given by the following formula:

$$\Delta_L(i) = \frac{l_p(i) - l_m(i))}{l_p(i)}. \quad (4)$$

This metric reflects the distance between the latency of each iteration from the measured and predicted schedule.

(2) Let the makespan of schedule σ be $ms(\sigma) = \max_j(ft(t_j))$. We define the makespan distance as

$$\Delta_M = \frac{ms(\sigma_p) - ms(\sigma_m)}{ms(\sigma_m)}. \quad (5)$$

In Figure 9 we show the measured distance in latency for various task lengths in milliseconds. When predicting the

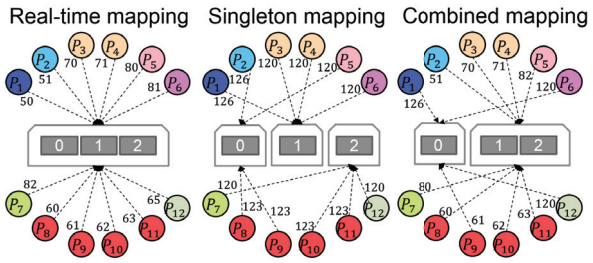

Figure 8: The mapping of the processes onto the platform.

singleton (purple) and the combined (green) mappings, we see that the distance increases as the task length decreases. This behaviour is explained by the used approximation: when approximating the progress by a linear function, the prediction errors increase when tasks approach the execution times of the time slicing granularity used by the scheduler. In our experiment the time slicing granularity is observed to be a constant four milliseconds. The real-time mapping (yellow) scheme does not exhibit this behaviour because time slicing is not used. As a consequence, the prediction errors are even smaller than in the other cases.

An important observation from the experiments is that the measured time a component is scheduled on the resources is constant across various mappings. This indicates that we can measure the nominal execution time of each task under any arbitrary mapping, even if the timing aspects of these mappings are unpredictable. We can infer a model from any measurement, and use this to predict the schedules of an alternative mapping.

Table I shows the results when we use a set of models inferred from one mapping to predict the schedule of the alternative mappings for a series of experiments with a task length of 24 milliseconds. We use the average of the absolute values of the distance for this table. The results show that the predicted values are dependent on the mapping that is being predicted, rather than the measured dataset that is used to predict the mapping. The observed prediction errors are very small, especially for the total makespan distances.

Used dataset	Predicted mapping	Δ_L [x100%]	Δ_M [x100%]
Real-time	Singleton	0.81	0.07
Real-time	Combined	1.52	0.17
Singleton	Real-time	0.18	0.05
Singleton	Combined	1.56	0.17
Combined	Real-time	0.28	0.07
Combined	Singleton	0.78	0.08

Table I: Experimental results for a task length of 24ms

VII. Related work

Existing work demonstrates that the use of affinity is an effective method to improve performance of applications

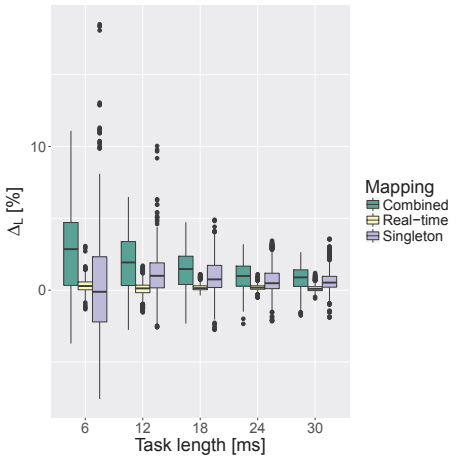

Figure 9: A boxplot of the observed errors for various task lengths.

of various types of software systems. Khatib-Astaneh et al. [8] have compared their previously proposed algorithms Earliest Feasible Deadline First (EFDF) and Feasible Least Laxity First (FLLF) with two well known algorithms EDF and global LLF. Their findings show that in particular core utilization has a massive improvement in especially EFDF as a result of using affinity of tasks. This study was motivated by the observation that under high load, preemption of tasks caused a share of 30% to 60% of the execution time to be spent on local memory and cache overhead. In another study on the topic of processor affinity, Muneeswari and Shunmuganathan [11] used 'hard'- and 'soft-affinity' for critical and non-critical tasks. They showed they could achieve maximum throughput for the critical tasks. This paper serves as motivation in developing a high-level analytical model to predict the performance of applications under the completely fair scheduler, under various affinity and niceness levels. More work on the topic has been done by Roberson [13], and Squillante and Lazowska [14]. Both works demonstrate the use of affinity for improving the scheduling of applications on multi-processors.

Two tools have been created to study the effect of Linux scheduling policies under various task distributions, *LinSched* [2] and *PRACTISE* [4]. Both tools work by simulating Linux functions such as $schedule()$ (*LinSched*) or $enqueue()$ and $dequeue()$ (*PRACTISE*). *LinSched* is a simulator tool which hooks on $scheduler\_tick()$ and $schedule()$ functions in Linux, simulating every individual time slice allocation. This makes the tool very adept at investigating the performance of various scheduling policies for various tasksets. *PRACTISE* is an emulator for scheduling policies similar as LinSched. During the execution of *PRACTISE*, it maintains a ready-queue for each processor being emulated and each cycle of the emulation consists of generating scheduling events at random, calling the corresponding scheduling functions and finally collecting statistics. Task activation is handled using the $pull()$ and $push()$ functions of Linux. *PRACTISE* provides a data structure for the use of a user of the tool for testing new algorithms.

Both *LinSched* and *PRACTISE* are tools to investigate the performance of scheduling policies on a low level and may be used to simulate CFS. However, the tools are not intended to analyze the performance of system applications, in particular task execution graphs. Our work aims to contribute to this aspect of CFS scheduling, by developing a high-level approximation of the completely fair scheduler in combination with real-time priorities.

Scheduling for time sensitive workloads on Linux operating systems has been studied before. In the work of Lelli et al. [10], a scheduling strategy *SCHED_DEADLINE* is used to improve the quality of service for time critical tasks. In this work, for each task a period and running time is provided such that the strategy can use this to meet deadlines more often. This approach works well when scheduling periodic tasks. In this work we are interested in improving response times for aperiodic task sets using the default scheduling policies (CFS and RT round robin), by finding an optimal mapping.

Separation of concerns is a pivotal point in designing a model for performance analysis. We use a Y-chart approach as introduced in [9]. A separation of concerns as proposed in the Y-chart approach keeps the applications and platform architecture separated by a mapping model which projects the application model onto a platform model. In most work, the Y-chart maps an application directly onto the platform. Unlike those works, we use a two-layered approach to map the application onto the platform, by first mapping tasks on processes (i.e. Linux resources) and subsequently mapping these processes are mapped onto the Linux compute platform. The advantage of our approach is that this two-layered mapping is capable to find also an optimal mapping of functions provided by each service. The Y-chart is used as a basis for the work from Hendriks et al. [5], which identifies various aspects of the modeling blueprint. This modeling blueprint represents the model of computation and differentiates between an untimed part (Y-chart layer with application, mapping and platform models), and a timed part (execution model layer, with task dynamics and resource dynamics). In our work, we use a similar approach and implicitly define an execution model layer that is representative of the Linux platform we consider, focusing on the timed aspect of the modelling blueprint. In our execution model layer, we also make use of a linear approximation of the progress of tasks when sharing a resource, however, unlike [5] we allow each task to have a weighted share of the resource.

Learning the task graph of a real-time system from its execution traces is accomplished in various ways. In Hendriks et al. [6], task graphs are heuristically learned from a schedule of a real execution and in Iegorov's work [7]

task precedence graphs are learned from (streaming) system traces. Compared to their work, we have more information available to construct a task graph: we trace remote procedure calls with exact dependency information such that our task graph is both sound (all dependencies in the model correspond to dependencies in the application) and complete (all dependencies in the application have a corresponding dependency in the model). This is in contrast to the work of Hendriks which may provide spurious dependencies or miss dependencies and the work of Iegorov which is an over-approximation of the dependencies. This allows us to construct a directed acyclic graph as the base for our model.

VIII. Conclusions

We developed a technique to transform trace data into a predictive analytical model for Linux-based service-oriented systems. With a linear progress approximation technique we can predict the schedule in a scalable manner. We showed that for real-time mappings, we can predict the schedule of an application with less than 2% of error. In singleton clusters with non real-time priorities we find that our model predicts within 5% when scheduling tasks with an execution time of more than three times the time slicing granularity with some outliers near 15% close to the time slicing granularity. When we combine both real-time priority clusters and singleton clusters the predictions are within 7%. The relative accuracy reduces as task times approach the time-slicing granularity.

We further showed that predictable execution of applications can be achieved under various constraints on the mappings. For mappings not satisfying these constraints, the Linux OS makes non-deterministic binding decisions, and the execution schedules of these applications therefore are also non-deterministic. This non-determinism may lead to performance issues due to missed deadlines of critical tasks of the application. We therefore recommend to limit mappings of time-critical processes to satisfy the partitioning and the constraints on the partition.

References

[1] Chapter 3 rpcgen programming guide . https://docs.oracle.com/cd/E19683-01/816-1435/6m7rrfn7f/index.html. Accessed: 23-4-2018.

[2] John M. Calandrino, Dan P. Baumberger, Tong Li, Jessica C. Young, and Scott Hahn. Linsched: The linux scheduler simulator. *Proceedings of the 21st International Conference on Parallel and Distributed Computing and Communications Systems, pages 171-176*, 2008.

[3] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *Proceedings of the OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Linux Symposium, 2006.

[4] Fabio Falzoi, Juri Lelli, and Giuseppe Lipari. Practise: a framework for performance analysis and testing of real-time multicore schedulers for the linux kernel. *Proceedings of the 8th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2012.

[5] Martijn Hendriks, Twan Basten, Jacques Verriet, Marco Brassé, and Lou Somers. A blueprint for system-level performance modeling of software-intensive embedded systems. *International Journal on Software Tools for Technology Transfer*, 18(1):21–40, Feb 2016.

[6] Martijn Hendriks, Jacques Verriet, Twan Basten, Bart Theelen, Marco Brassé, and Lou Somers. Analyzing execution traces: critical-path analysis and distance analysis. *International Journal on Software Tools for Technology Transfer*, 19(4):487–510, Aug 2017.

[7] Olog Iegorov and Sebastian Fischmeister. Mining task precedence graphs from real-time embedded system traces. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), Proceedings of the 24th Conference on*. IEEE, 2018.

[8] N. Khatib-Astaneh, V. Salmani, H.T. Yazdi, and M. Salmani. Investigating the effect of processor affinity on uniform parallel machine scheduling. In *Proceedings of the 2009 International Conference on Application of Information and Communication Technologies*, pages 1–5, Oct 2009.

[9] Bart Kienhuis, Ed Deprettere, Kees Vissers, Pieter van der Wolf, Paul Lieverse, and Edward Lee. Y-chart methodology and models of computation and architecture. *37th Design Automation Conference*, 2000.

[10] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 6 2016.

[11] G Muneeswari and KL Shunmuganathan. Agent based load balancing scheme using affinity processor scheduling for multicore architectures. *WSEAS Transactions on Computers*, 10(8):247–258, 2011.

[12] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[13] Jeff Roberson. Ule: a modern scheduler for freebsd. USENIX–The Advanced Computing Systems Association, 2003.

[14] M.S. Squillante and E.D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, Feb 1993.

[15] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 62–75, Dec 2015.

[16] Liang-Jie Zhang, Jia Zhang, and Hong Cai. *Service-Oriented Architecture*, pages 89–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.